BOLDFIELD LIMITED
COMPUTING

ASSEMBLER/
DIS-ASSEMBLER

Users Manual

This program is a fast and powerful *Z80* ASSEMBLER / DIS-ASSEMBLER written for the Jupiter ACE computer. Unlike similar programs for other computers this package actually generates FORTH primitives on a par with the Ace's own ROM words.

Although sophisticated in operation, this program is exceptionally simple to use, and is an invaluable aid to the writing of fast ACE software.

The supplied cassette also contains three demonstration listings.

INTRODUCTION

At the heart of the ACE there is a CPU (Central Processor Unit) which is equivalent to the Zilog Z80A. All this understands is machine code instructions - normally supplied by the ROM (Read Only Memory), which translates programs written in the higher level language of FORTH.

It is obvious that to write your software in the language directly used by the CPU would yield benefits of speed and capability, as the computer would not need to undergo the translation process. However, it is very tiresome to write programs using just a string of numbers, especially when it comes to finding bugs ! Consequently, serious programmers adopt a half way house known as Assembly language.

Each machine code instruction is given a meaningful mnemonic so that humans can understand and remember them, and an ASSEMBLER program, such as this one, converts the mnemonics to machine code numbers. The reverse procedure is naturally performed by a DIS-ASSEMBLER.

This manual assumes that the user is familiar with Z80 mnemonics, and he is referred to Appendix A in the Jupiter ACE user's manual for a list of most of them, and to Chapter 25 which deals with machine code generally.

LOADING INSTRUCTIONS

The program uses approximately 11 kilobytes of dictionary space.

To load the program enter: load a1

To load the demonstration listings enter: load examples

Here is a listing of an assembler program (comments in curly brackets):

```
CODE DEMO    {- - }
      \ ld a,N     \ ASCII A )
      \ Id hl,NN   \ 9216 ))
      \  ld(hl),a \
      \  jp(iy)    \
ENDC
```

This can be typed in line by line or all at once on the ACE, and is an example of the assembler in immediate mode ie. no record is made of the listing. The setting up of text files of your assembler listings will be explained later.

CODE and ENDC are special words that start and terminate named sections of code.

The result of typing the above would to be define a new FORTH primitive at the top of the dictionary by the name of DEMO, on a par with words like DUP, DROP etc. It would load the accumulator with the number 65 (ascii A), the HL registers with the address of the start of the display file for the television screen, output the contents of the accumulator indirectly via the HL registers and return to FORTH. To execute the word simply type DEMO - there is no need to use CALL. The letter A will appear at the top left of the screen. DEMO needs no supporting words and can be saved by itself.

The first point to notice is that all the instructions appearing between '\' symbols are of the form found at the back of the Jupiter ACE manual ie. instructions that require numeric parameters are written with 'N' 'NN' 'DIS' and 'D' in their place, and the numeric parameters themselves follow after the second '\' and are enclosed with ')' or '))' for one and two byte parameters respectively. The reason for maintaining this format is that it allows great flexibility in forming parameters - indeed, FORTH expressions of any degree of complexity are possible, so long as the resultant number is compatible with its instruction.

'D' is the Displacement to be found in ix/iy instructions such as :

```
        \ ld a,(ix+D)\ -3 )          where -3 is its one byte parameter
```

Now, in the case of the following :

```
       \ ld(ix+D),N \ 120 ) H F7 )
```

the two data bytes must appear in the same order as they do in the instruction ie. 122 is the displacement, and F7 corresponds to 'N'.

This example also illustrates the way to change the number system. The default is decimal, but the words H, B, and D change the system to Hex, Binary and Decimal respectively. The change of base only lasts until the next use of ')' '))', or '\'. Additionally, any other number base can be used simply by typing 'n BASE C!' where n is the base number.

'DIS' is the relative DISplacement to be found in relative jump statements such as :

```
      \ jr DIS  \ 1 )
      \ xor a   \
      \ Id a,b  \
```

In this example '1' is the displacement and would have the effect of skipping out 'xor a' when running. However, it is far more likely that you would use Labelled jumps such as :

```
      \ jr DIS  \ ~ A
      \ xor a   \
£ A   \ ld a,b  \
```

Any instruction may be assigned a label. However, the label declaration must occur immediately before its instruction, with '£' as the declaring word' which must be followed by a single letter. Lower and upper case letters are allowed, giving a maximum of 52 different labels.

Note that no ')' should appear after a reference to a label, and that the word '~' must precede the letter which makes reference to a labelled instruction in relative jumps.

Another addressing mode is that of Absolute Addressing where the actual address implied is stated specifically eg.:

```
         \ jp NN      \ H 04B9 ))
         \ call NN    \ H 04BB ))
{fe455}  \ jp(iy)     \
{04BB}   \ bit7,(hl)  \
         Ret
```

Now, using labelled jumps this could be rewritten :

```
         \ jp NN      \ ↑ A
         \ call NN    \ ↑ B
£ A      \ jp(iy)     \
£ B      \ bit7,(hl)  \
         \ ret
```

The procedure is the same as in relative addressing, except the symbol '↑' replaces '~' when referring to a labelled instruction. Its not only 'jp' and 'call' that can use '↑' to good effect - the following might be referring to a buffer area within the program labelled 'D' :

```
         \ ld a,(NN) \ ↑ D    again note the absence of '))' after '↑ D'
```

It is always best to try and write code without recourse to the use of absolute addressing' since such code is relocatable and is not tied to an area of memory originally intended for it. However' when its use is unavoidable, it is necessary that you know the address that will correspond to the start of code. This is known as the ORiGin. If labelled addressing is to be used in such code the system variable ORG must be assigned this address AFTER the code name has been declared but BEFORE the start of instructions eg.:

```
CODE XYZ          H    5C00 ORG !
      \ di             \
      \ call NN        \ ↑ A
      \ iP(iy)         \
£ A   \ ei             \
      \ ret            \
ENDC
```

Dis-assembling the above would reveal that the address substituted for '↑ A' was 5C06, which is what it will be when the code is lying in the area of memory intended for it.

To save you the bother of calculating what the code ORiGin for a new word will be when using absolute label references, this Assembler uses another special word called ADR (which you may find handy to SAVE on it's own). It works like this - a new word SAVEd without ORG is LOADed to the dictionary in the location where it is to be used. To make it compatible with this location you LOAD ADR and enter : ADR 'name' FORGET ADR.
The new word (called 'name' in this case), will now execute correctly.
Should you subsequently use REDEFINE, you may have moved the position of the new word, and if so, it will need another ADR adjustment.

Of course, all this hassle is avoided if you can write your programs without using absolute addressing !

INSTRUCTION FORMAT

All user words (£, ↑, ~ etc.) must be followed by not less than one space, except the opening '\' which must be separated from its assembler mnemonic by one space only. Spaces do not appear between the first and last letters of an instruction unless the resultant form is ambiguous :

```
eg.   \ bit3,(ix+D)   \
      \ rst16         \
      \ ld(hl),a      \       have no spaces at all
```

whereas

```
      \ ld a,b        \
      \ jr nc,DIS     \
      \ jp NN         \       would be ambiguous without the spaces
```

The instruction terminator '\' may be preceded by any number of spaces or none at all. Code need not be entered on separate lines as this manual shows, but can instead be entered as a continuous stream, so long as the spacing convention is followed.

All Z80 mnemonics are entered in lower case, except 'N', 'NN', 'DIS' and 'D' which correspond to data bytes.

MACRO FACILITY

A macro is a section of machine code which is used several times over, rather like a subroutine in a BASIC program. However, it differs from a subroutine in that its code is inserted into the body of the machine code at each point that it is required. The advantage of a Macro lies in the increased speed of execution resulting from the lack of subroutine calls.

Below is a trivial example making use of the defining word MACRO to generate a named section of code which is used twice in the following primitive definition :

```
MACRO NEWL
      \ ld a,N\13 ) \ rst8 \
ENDC

CODE AB
      NEWL \ ld a,N\ascii A ) \ rst8\
      NEWL \ ld a,N\ascii B ) \ rst8\
      \ jp(iy)\
ENDC
```

If you LIST NEWL you will see it is simply defined as a colon definition containing the machine code bytes as 16 bit literals. Execution of NEWL (as a normal colon defined word in FORTH) puts the bytes on the stack.

Note that '↑' may not be used within a Macro definition.

DIRECT MACHINE CODE COMPILATION

It is possible to use the Assembler in a simple mode, where you enter the machine code bytes (in any base) directly. The example here sets up the word '+!' (n, addr --) which when executed increments the 16 bit contents of 'addr' by 'n'.

```
CODE +!
      H     DF D9 DF 05 D9 C1 EB 5E
            23 56 E8 09 E8 72 2B 73
            FD E9
ENDC
```

Note the use of 'H' to set the number base to Hex; and always write the hex numbers 8 and D as OB and 0D to avoid confusion with the other special words that affect the number base.

Machine code and assembler instructions can be freely mixed, and by doing so buffer areas and lists of data can be included in a program. It is not necessary to follow unsigned 8 bit numbers by ')', but signed 8 bit numbers and all 16 bit numbers must be followed by ')' and '))' respectively.

The following word CHARS prints the first ten letters of the alphabet read from a data list :

```
CODE CHARS   HERE ORG !
      \ ld hl,NN     \↑ A
      \ ld b,N       \ 10 )
£ B   \ ld a,(hl)    \
      \ rst8         \
      \ inc hl       \
      \ djnz DIS     \ ~S
      \ jp(iy)       \
£ A   { DATA LIST    }
      65 66 67 68 69
      70 71 72 73 74 \\
ENDC
```

Normally, is used to label assembler instructions, but it may also label data lists and macros, as long as that section of code is immediately followed by the word '\\' . Eg: £ A NEWL\\

To make buffer areas in a program, include the appropriate number of zeros, or alternatively define a word such as BUFFER :

```
: BUFFER ( n--n zeros ) 0 DO 0 LOOP ;
```

this could be used to reserve, say, 100 bytes at the end of a section of code and to label the first byte as 'A' :

```
\ jP(iY) \
£ A 100 BUFFER \\
```

CALLING FORTH FROM MACHINE CODE

This is how to include lists of FORTH compilation addresses within machine code which will run as though from a colon definition. Engage the FORTH inner interpreter using the instruction 'call 04B9H' and then include as the last compilation address '1A0EH' to revert to machine code. The example below would square the contents of the DE register and then divide it by three:

```
\ rst16          \   { TOS ← DE }
\ call NN        \ H 0489 ))
                       { FORTH CA's }
                       FIND DUP ))
                       FIND * ))
                       FIND LITERAL 2+ @ )) 3 ))
                       FIND / ))
                       H 1A0E ))
\ rst24          \   { DE ← TOS }
```

Note how the compiling word LITERAL stores its run-time compilation address 2 bytes on from the address returned by FIND LITERAL.

ERROR MESSAGES

If the Assembler does not recognize an instruction, it will void the entry and print INVALID in inverse video immediately after the unacceptable listing. Previous instructions are unaffected, and you only need retype the part which is erroneous.

Another type of error that can occur is not detected until execution of ENDC. This is caused by reference to an undeclared label, and will produce the message 'LABEL UNDECLARED £ x ?' where x is the label in question. In which case you will need to FORGET the offending word from the dictionary and try again !

SAVING WORDS ON TAPE

To SAVE a newly defined word at the top of the dictionary, first REDEFINE the lowest word in the dictionary (after FORTH), and then FORGET the word above it to leave your code word as the only non-ROM item. You can now SAVE your word on its own.

TEXT FILES

For programs of more than a few lines, you may want to make a permanent record of your assembly listings for later reference or to enable changes to be made without having to retype the whole lot in again.

The first thing to do, before you load the Assembler program, is to define a file called, say, TEXT1

```
: TEXT1
   ." "
;
```

Next simply edit your assembler listings (called the Source Code) in between the quotation marks of the FORTH word TEXT1. You may wish to follow the neat one-instruction-per-line format used here :

```
: TEXT1
."
CODE DEMO { - - }
     \ ld a,N    \ ASCII A )
     \ ld hl,NN \ 9216 ))
     \ ld(hl),a \
     \ jp(iy)    \
ENDC "
;
```

In this listing the first column on the TV screen was left blank, and a further four spaces were left before '`' to leave room for any future labels. This layout makes LIST work better and maintains the neat appearance after changes.

Now, the modified word TEXT1 can be SAVEd and VERIFIED in the normal way. Without resetting the computer, LOAD the Assembler program. Enter LST TEXT1 to produce a listing of the assembler instructions starting at line 4. To run this listing through the assembler enter READ. READ is a command that raises the input buffer to line 3, and has the same effect on the displayed instructions as if you had just typed them in on the keyboard.

Another special word, 'V' ,is similar to 'VLIST' only it also shows how many bytes are used before displaying the current dictionary. So, if you had just READ the TEXT1 file and entered 'V' you would see DEMO as the top word.

If you had made any mistakes in the TEXT1 listing, it could not be assembled completely, and the partially formed word would need FORGETting from the dictionary, and then corrected by EDITing as if it was an ordinary FORTH colon definition.

In the case of programs too large to fit in one file (about 19 lines), files can be chain-linked together by having the last instruction in TEXT1 as 'LST TEXT2 READ' where TEXT2 is a continuation file in the same form as TEXT1. Any number of files can be linked in this way, with ENDC as the last instruction in the last file. For very large programs, where memory space is at a premium, you would have to forgo the neat layout and use multiple instructions per line instead !

NOTES ON THE ASSEMBLER

Instructions, macros, data bytes etc. are not compiled immediately into the dictionary, but are left on the stack until ENDC (with letter labels embedded into the higher order byte of each stack entry). ENDC uses the word 'DEPTH' to determine the number of machine code bytes generated, makes a pass over the stack processing label references, and then compiles the low order bytes of each stack entry into the parameter field of the word created by CODE.

Since CODE puts the ACE in FAST mode, no checks are made for stack overflow etc., and consequently, there is a risk of crashing if you have made mistakes in your listings. For added safety you may type SLOW immediately after 'CODE name'.

The ACE manual does not list Z80 instructions using iz (where iz=ix/iy), but in general, any instruction with hl as a parameter has its equivalent form, although '(hl)' must be replaced by '(iz+D)' . An exception is ex de,hl which has no equivalent.

Furthermore, on this Assembler :

XXX a,(iz+D) is entered XXX(iz+D), where XXX=add/adc/sbc

DEMONSTRATION LISTINGS ON THE CASSETTE

The dictionary file containing the demonstration listings is called 'examples'. It contains three listings, EX1, EX2 and EX3.

To assemble the first one, enter LST EX1 (you may like to compare the screen with LIST EX1, to see how LST works) followed by READ. A new word 'FILL' will now be found in the dictionary. Many FORTH systems support FILL (addr,N,CHAR - -). It fills an area of memory with the character specified. For example, 9216 736 ASCII X FILL covers the TV screen with X's. You might like to try the following colon definition :

```
: RUN 256 0 DO 9216 736 I FILL 3910 0 DO LOOP LOOP
```

The second example is not an executable program and only serves to illustrate some of the Assembler's facilities. View the listings by entering LST EX2 and LST EX2A . Note how EX2 is chain-linked to the continuation file EX2A on the last line, and that ENDC only occurs on the last line of the last file. Also, CONSTANT OFILE declared in the listing, and referred to twice, must be declared before the word CODE.

EX3, after assembling, produces the word SIREN, which if entered makes the ACE beep unit produce a siren noise - use BREAK to stop the din !
Now you can use this word to demonstrate AOR. First satisfy yourself (by doing a LST EX3) that SIREN contains absolute label references and does not specify a code ORiGin. Next relocate SIREN by say, REDEFINE EX2A, and wipe out the unseen original SIREN in the dictionary by say, EDITing EX2 (without making any changes - just to produce a duplicate). Now, you have a brave decision to make - if you execute SIREN in it's new position the computer will crash, or you could instead enter: ADR SIREN , and all will be well !

USING THE DIS-ASSEMBLER

To engage the Dis-Assembler, simply precede the special word DSM with the address from which you wish to dis-assemble. For example : 0 DSM will commence at address location 0 in the ACE's ROM. The listing continues only while you hold the ENTER key down, and can be stopped and restarted as you wish. With the ENTER key released the program is in interpret mode, and any commands typed in will be executed on the next use of ENTER. This feature is used to change number bases, or modify the next address etc.
Since DSM leaves the next address on the stack, simply overlay. it with a different one to cause dis-assembly to continue from some other location.

To exit from the Dis-Assembler, enter V, which also executes VLIST.

The screen format is as.follows

```
Col   0 to   1  ... the 8 bit contents of the current address in Hex
Col   3 to  13  ... the Z80 mnemonic in generalised form, alongside the first
                    byte of each instruction
Col 14 to   21  ... the absolute address corresponding to the destination of
                    relative jumps, in Hex, enclosed in square brackets
Col 23 to 26    ... the current address, in Hex
Col 31          ... the character representation of the current address
```

Although mnemonics appear in generalised form, as found in the ACE manual, no information is lost since the data bytes are indicated by being followed by ')'. The more significant byte of an address appears last as is usual in Z82 code.

ERROR MESSAGES

Only one type of error can arise during dis-assembly, and this is indicated by the message ILLEGAL CODE SEQUENCE. This means that the code is not Z80 instructions by virtue of the ordering of the code bytes. If this occurs in ROM it is most probably a data list or FORTH compilation addresses.

GLOSSARY OF USER WORDS

£	(- -)	Followed by space,letter. Labels a Z80 instruction during assembly
~	(- - n)	Followed by space,letter. Makes reference to a labeled instruction from a relative jump
↑	(- - n,n)	Followed by space letter. Makes reference to a labeled instruction using absolute addressing mode
{	(- -)	Followed by a string and terminated by '}' allows comments to be inserted in listings
\	(- - nos)	Followed by Z80 mnemonic and terminated by '\', converts mnemonic to machine code bytes
\\	(code bytes - - labelled code bytes)	Must follow data bytes or MACROs that have been labelled by
)	(n - - n)	Encloses 8 bit data parameter into preceding instruction
))	(n - - n,n)	similar, but for 16 bit data parameter
B	(- -)	Changes number base to Binary
D	(- -)	Changes number base to Decimal
H	(- -)	Changes number base to Hexadecimal
V	(- -)	Terminates dis-assembling, prints number of bytes used and performs a VLIST
ADR	(- -)	Followed by name of primitive, adjusts absolute addressing to be compatible with current dictionary location
DSM	(- -)	Preceded by address, engages Dis-assembler from address
LST	(- -)	Followed by name of text file, correctly formats text on screen ready for READ
READ	(- -)	Raises input buffer to line 3 enabling a listed assembler program to be interpreted and assembled
ORG	(- - n)	Variable, must be assigned address of code origin when '↑' is used - unless ADR is intended to be used
CODE	(- -)	Followed by name of new code word, sets up header in the dictionary and primes the Assembler
MACRO	(- -)	Similar to CODE except causes ENDC to compile machine code bytes into dictionary as 16 bit literals. Code field of header is as for a colon definition
ENDC	(- -)	Terminates Assembler operations, processes labelled jumps and encloses code bytes into the dictionary